Service without Servers

Chris Maeda Brian N. Bershad

August 1993
CMU-CS-93-144

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the Fourth Workshop on Workstation Operating Systems. October 14-15, 1993.

Abstract

We propose a new style of operating system architecture appropriate for microkernel-based operating systems: services are implemented as a combination of shared libraries and dedicated server processes. Shared libraries implement performance critical portions of each system service, while dedicated servers implement the parts of each service that do not require high performance or that are difficult to implement in an application. Our initial experiments show that this approach to operating system structure can yield performance that is comparable to monolithic kernel systems while retaining all the modularity advantages of microkernel technology. Since services reside in libraries, an application is free to use the library that is most appropriate. This approach can even yield better performance than monolithic kernel systems by allowing the shared libraries to be closely coupled with the applications, thereby exploiting application-specific knowledge in policy decisions.

This research was sponsored in part by the Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by ARPA/CMO under Contract MDA972-90-C-0035, the Xerox Corporation, and Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award. Maeda was partially supported by a National Science Foundation Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, Xerox Corporation, Digital Equipment Corporation, the National Science Foundation, or the U.S. Government.

Keywords: Operating Systems, Microkernel, Network communication, File organization

1. Introduction

In the past few years, there has been dramatic growth in the number and quality of microkernels. As of this writing, several commercial operating systems based on microkernel technology exist or are under development [Phelan et al. 93, Hildebrand 92, Rozier et al. 92, Zajcew et al. 93]. Current practice is to structure a microkernel operating system as one or more server processes that collectively implement the operating system services [Golub et al. 90, Julin et al. 91, Rozier et al. 92, Khalidi & Nelson 93, Hildebrand 92]. This approach implicitly models the operating system as a distributed system where services reside in "remote" server processes that happen to be on the same machine. However, the communication overhead incurred when contacting these servers can result in poor performance. In latency-sensitive applications like those requiring high-speed networking, the extra communication costs for applications to communicate with "remote" server processes is unacceptable because the latency for an interprocess RPC is comparable to the network round-trip latency [Brustoloni & Bershad 93, Draves et al. 91].

In this position paper, we propose a new style of operating system architecture appropriate for microkernel-based operating systems: services are implemented as a combination of shared libraries and dedicated server processes. Shared libraries implement performance critical portions of each system service, while dedicated servers implement the parts of each service that do not require high performance or that are difficult to implement in an application. Dedicated servers might be used, for example, to manage shared state that must persist across process lifetimes or to implement high-level abstractions that are difficult or impossible to provide in a library.

Our initial experiments show that this approach to operating system structure can yield performance that is comparable to monolithic kernel systems while retaining all the modularity advantages that led industry to adopt microkernel technology in the first place. Since services reside in libraries, an application is free to use the library that is most appropriate. This approach can even yield better performance than monolithic kernel systems by allowing the shared libraries to be closely coupled with the applications, thereby exploiting application-specific knowledge in policy decisions.

In the next section, we present our approach to structuring system services and discuss the role of the kernel, servers, and application-level libraries. In Section 3 we describe how we applied our approach to the implementation of a networking service (a more complete description can be found in [Maeda & Bershad 93]). In Section 4 we suggest how our approach may be applied to a filesystem service, and discuss the potential benefits of doing so. We summarize in Section 5.

2. Service structure

Operating systems generally perform two functions: they allocate machine resources, such as physical memory, processors, and I/O capacity, and they provide high-level abstractions like filesystems, processes, and I/O channels. In our model, the kernel is simply a global resource scheduler. Services external to the kernel provide abstractions and a means for applications to acquire and use resources. The implementation of a service is split into global and application-specific parts that reside in servers and shared libraries, respectively. By splitting the implementation in this way, applications can make more effective use of their resources without forfeiting security or generality.

Services acquire resources from the kernel and manage them using a global server or application-specific libraries. Global servers can be used to manage shared abstractions or large blocks of resources delegated by the kernel, such as a disk partition. The application specific libraries provide information to the kernel about special resource requirements and manage resources dedicated to the application, such as a single network connection. The kernel in such a system has the following requirements:

- a global resource scheduling policy.

- adequate means to enforce resource scheduling decisions (protection).
- a way to inform services about resource scheduling decisions.
- a way for services to provide hints to the kernel to influence resource scheduling decisions.

There are two performance benefits to our model. The first is that the shared library can exploit application-specific knowledge in managing resources allocated by the kernel. The second benefit of our model is that performance-critical parts of the service can be implemented in the shared library, thereby avoiding the extra communication overhead associated with a remote server process.

Some examples

The operating systems community has been "dabbling" with the resource manager model for some time now. Early personal computer operating systems [Redell et al. 80, Moon 91] ran all system services and applications in a single address space, which enabled applications to be tightly coupled with the operating system. However, these systems provided no protection against rogue or buggy applications that crash the system or use the hardware to attack other systems.

Other work spans file systems [Rees et al. 86, Bershad & Pinkerton 88], scheduling [Anderson et al. 92], communication [Bershad et al. 91], and user-level memory management [McNamee & Armstrong 90, Harty & Cheriton 92, Sechrest & Park 91, Krueger et al. 93]. The work in extensible filesystems permits applications to extend the semantics of files on a per-file basis. However, this work still leaves all resource scheduling decisions to the operating system. With scheduler activations, the kernel globally allocates processors to applications and informs them when their processor allocation changes. The applications provide hints about when changes in processor allocation would be useful and use the processor allocation information to implement a high-level threads library. URPC is an IPC library that uses shared memory to implement low-latency IPC. The library relies on the kernel's scheduler to perform processor allocation in response to outstanding messages. Implementations of user-level memory management permit applications to determine the page replacement policy for their virtual memory. The kernel allocates physical memory to applications while the applications determine virtual to physical mappings. (In contrast, the Mach External Pager [Young 89] simply enables applications to implement backing store for parts of their virtual memory.)

3. A networking service

In this section, we describe an implementation of networking protocols that runs as a library-level service, and that relies on a central server for a few critical operations. In our networking service, a library implements a complete TCP/IP and UDP/IP stack that communicates directly with an in-kernel Ethernet device driver. Incoming packets are demultiplexed to application address spaces using the packet filter, a protocol-independent packet demultiplexing facility [Mogul et al. 87, Yuhara et al. 94]. The library takes raw Ethernet packets from the kernel and does all protocol stack processing before handing the data to the application. Similarly, outgoing data is formatted into TCP/IP or UDP/IP messages before being sent to the in-kernel Ethernet driver. The proper level of distrust is maintained because the packet filter ensures that applications only see the packets that they are supposed to see. A similar mechanism could be applied to outgoing packets to ensure that applications only send packets that they are allowed to send.

The library is linked with applications and cooperates with a server process to manage host-level state such as routing tables and to support the BSD Sockets API (see Figure 1). The Sockets API is difficult to emulate in an application-level library because network sessions are represented as file descriptors which have complex semantics due to system calls such as **fork** and **select**. Two techniques, ***session state migration*** and ***co-management of abstractions***, are instrumental in emulating the complex semantics of Unix file descriptors.

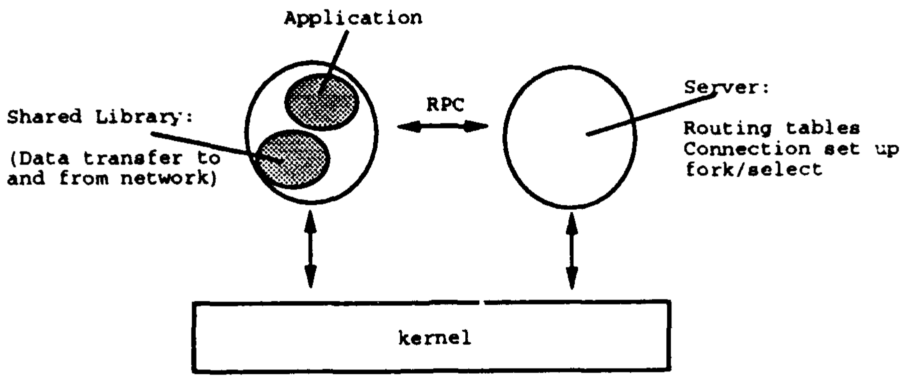

Figure 1: Schematic of a networking service implementation where critical-path functionality is implemented by libraries in the application's address space. A server process manages shared databases, handles connection set up, and implements high-level abstractions.

Session state migration is used to set up sessions and to enable sharing. The session state for a TCP connection, for example, consists of the connection state variables ([Postel 81]), and any data buffered by the protocol stack. The server process handles the TCP connection setup protocol and migrates the established TCP session to the application during an `accept` call. The protocol stack in the application and the in-kernel device driver handle all subsequent data transfer; no interaction with the server is required. When an application does a `fork` system call, the parent and child processes of the fork must each have a file descriptor that refers to the same network session. Before the fork, the network session state is migrated *back* to the server process so that it may be shared.

Co-management of abstractions is used to emulate the `select` system call. The operating system knows about file descriptors that are managed by applications and exports an interface by which the applications can inform the operating system when the status of a file descriptor changes. When the library learns that a file descriptor has changed status, it informs the operating system which forces any blocked `select` calls to return.

The global resource managed by the kernel is network capacity. When an application acquires a packet filter port, it acquires a portion of network capacity on which it can apply a protocol. In the current implementation, this resource allocation aspect is implicit; we assume that Ethernet bandwidth is infinite. However, the kernel could explicitly allocate network resources if applications specified a quality of service (QOS) [Kurose 93] at session establishment time, and if the kernel enforced QOS by penalizing applications that exceed their limits.

The performance of our system is comparable to native in-kernel implementations. Between two DECstation 5000/200's running in single-user mode on a private Ethernet, the round-trip latency for 1 byte UDP messages is 1.50 ms in our system and 1.45 ms for the Mach 2.5 integrated kernel. Round-trip latency for 1 byte TCP messages is 1.40 ms in Mach 2.5 and 1.75 ms in our system. For TCP throughput, the Mach 2.5 kernel achieves 1070 kilobytes/second while our system achieves 995 kilobytes/second. Our library lags a few percent behind 2.5 because we copy incoming packets (including protocol headers) one extra time in our system. In contrast, both implementations substantially outperform the Mach 3.0 Unix Server, where the round-trip latency for 1 byte messages is 3.61 ms for UDP, 3.64 ms for TCP, and where TCP throughput is 740 kilobytes/second.

Once we have a user-level implementation, we can achieve further performance gains by more tightly coupling the application with the protocol implementation. For example, we can change the API to return a buffer from a socket call, rather than filling one in, eliminating two data copies during round-trip operations. UDP round-trip latency drops to 1.46 ms and TCP latency drops to 1.72 ms. TCP throughput only improves by 1% (to 1002 kilobytes/second) because the copies eliminated by this change are not the critical path for high throughput.

4. A proposed filesystem service

At first glance, it is not clear how application-specific libraries can provide any performance improvement for filesystems. Since many applications share the same file system, the filesystem metadata should be managed by a single server to which the kernel has delegated responsibility for the disk partition. Furthermore, disks are slow enough compared to the cost of an interprocess RPC that there is little additional overhead involved with accessing data through a dedicated file server process [Bershad 92]. However, given current trends in CPU, memory system, and disk performance, applications will require a fast path to a disk block cache which rarely misses in order to perform well. By "fast," we mean that there isn't time to even copy the data out of the buffer cache into the client's address space [Ousterhout 90], let alone fetch it from disk. Consequently, the data must effectively be cached in the client's address space before it is accessed.

One way, then, to apply our resource management methodology to a file system service is to have a server that manages the disk, along with a shared library that implements a distributed buffer cache. Instead of having a buffer cache that resides in the address space of a single server, the buffer cache pages can be mapped into the address space of the application that most likely to use them. This approach means that an application can access data in the buffer cache with a simple procedure call and without having to copy the data out of the cache.

While this approach uses the same *mechanism* as memory-mapped files, the *policy* is completely different. Memory-mapped files use a reactive policy where file data is not brought in from disk and mapped into the application's virtual address space until it is referenced, forcing the application to wait. Future applications, such as those in databases, multimedia, and scientific computing, will require a more proactive policy where data is present in physical memory before it is first referenced.

Filesystem performance can benefit from application-specific information in two ways. The application can provide hints about future usage to the filesystem server to help it schedule disk traffic [Patterson et al. 93]. This will result in more effective prefetching policies and lower buffer cache miss rates. An effective prefetching policy will also move virtual memory remapping operations off the critical path since disk blocks will already be mapped into the application address space when they are needed. In addition, the application can tell the kernel about how it will use the buffer cache so that the kernel can make informed decisions about physical memory allocation [Stonebraker 81].

5. Summary

One way for microkernel operating systems to match the performance of integrated kernel systems is to implement services as a combination of application-level shared libraries and dedicated server processes. We have implemented a networking service in this "distributed" fashion that matches the performance of an integrated kernel and we have demonstrated that tighter coupling between the application and the library can result in even better performance. We expect that other operating system services can be designed this way; library-based implementations of scheduling, IPC, and memory management have been reported in the literature, and we have described a way that this approach could be applied to file systems.

An open problem is how to design an infrastructure that enables "distributed" implementations of operating system services. The most important part of this infrastructure is a kernel that informs applications about its resource scheduling decisions and that can use hints about application requirements in its resource scheduling decisions.

References

[Anderson et al. 92] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 9(1), February 1992.

[Bershad & Pinkerton 88] Bershad, B. N. and Pinkerton, C. B. Watchdogs: Extending the UNIX File System. In *Proceedings of the USENIX Winter Conference*, pages 267–275, February 1988.

[Bershad 92] Bershad, B. N. The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.

[Bershad et al. 91] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[Brustoloni & Bershad 93] Brustoloni, J. C. and Bershad, B. N. Protocol Processing for Low-Latency High-Bandwidth Networking. TR CMU-CS-93-132, School of Computer Science, Carnegie Mellon University, March 1993.

[Draves et al. 91] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, October 1991.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Harty & Cheriton 92] Harty, K. and Cheriton, D. R. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedingsof the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 187–197, 1992.

[Hildebrand 92] Hildebrand, D. An Architectural Overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, 1992.

[Julin et al. 91] Julin, D. P., Chew, J. J., Stevenson, J. M., Guedes, P., Neves, P., and Roy, P. Generalized Emulation Services for Mach 3.0 Overview, Experiences and Current Status. In *Proceedings of the Usenix Mach Symposium*, pages 13–26, 1991.

[Khalidi & Nelson 93] Khalidi, Y. A. and Nelson, M. N. An Implementation of UNIX on an Object-oriented Operating System. In *Proceedings of the Winter 1993 USENIX Conference*, pages 469–479, 1993.

[Krueger et al. 93] Krueger, K., Loftesness, D., Vahdat, A., and Anderson, T. Tools for the Development of Application-Specific Virtual Memory Management. Technical Report CSD-93-740, UC Berkeley, 1993.

[Kurose 93] Kurose, J. Open Issues and Challenges in Providing Quality of Service Guarantees in High-Speed Networks. *Computer Communication Review*, 23(1):6–15, January 1993.

[Maeda & Bershad 93] Maeda, C. and Bershad, B. N. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the 14th Symposium on Operating Systems Principles*, December 1993. To appear.

[McNamee & Armstrong 90] McNamee, D. and Armstrong, K. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17–29, 1990.

[Mogul et al. 87] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 39–51. ACM, November 1987.

[Moon 91] Moon, D. A. Genera Retrospective. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems (IWOOOS-91)*, pages 2–8, October 1991.

[Ousterhout 90] Ousterhout, J. K. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Usenix 1990 Summer Conference*, pages 247–256, June 1990.

[Patterson et al. 93] Patterson, R. H., Gibson, G. A., and Satyanarayanan, M. A Status Report on Research in Transparent Informed Prefetching. *Operating Systems Review*, 27(2):21–34, April 1993.

[Phelan et al. 93] Phelan, J. M., Arendt, J., and Ormsby, G. R. An OS/2 Personality on Mach. In *Proceedings of the Usenix Mach III Symposium*, pages 191–201, 1993.

[Postel 81] Postel, J. Transmission Control Protocol. Request for Comments 793, USC Information Sciences Institute, September 1981.

[Redell et al. 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Rees et al. 86] Rees, J., Levine, P. H., Mishkin, N., and Leach, P. J. An Extensible I/O System. In *USENIX Association Summer Conference Proceedings*, pages 114–125, June 1986.

[Rozier et al. 92] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Léonard, P., and Neuhauser, W. Overview of the Chorus Distributed Operating System. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69, 1992.

[Sechrest & Park 91] Sechrest, S. and Park, Y. User-Level Physical Memory Management for Mach. In *Proceedings of the Usenix Mach Symposium*, pages 189–199, 1991.

[Stonebraker 81] Stonebraker, M. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[Young 89] Young, M. W. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Technical Report CMU-CS-89-202, Carnegie Mellon University, November 1989.

[Yuhara et al. 94] Yuhara, M., Bershad, B. N., Maeda, C., and Moss, J. E. B. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.

[Zajcew et al. 93] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., and Netterwala, D. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*, pages 449–468, 1993.